**09/09/25 – BACKEND DE APLICACIONES**

Programación funcional: paradigma que tiene una base matemática compleja y que existe en pocos lenguajes puros (incluyendo lenguajes de uso cotidiano como Java, C#, JavaScript, etc.). Estos lenguajes dan la posibilidad de trabajar con un tipo de dato llamado función. En la programación funcional, las variables se almacenan código fuente, es decir, conjuntos de instrucciones del lenguaje. Por lo tanto, si una variable se declara una función, es posible ejecutar el código que esta contiene. Y dado que son variables pueden cambiar su contenido, por lo tanto, si en algún momento se asigna un código diferente a la variable y se ejecuta su contenido nuevamente y se llevará una ejecución distinta. Siempre se ejecutará el conjunto de instrucciones que este almacenado en la variable en ese momento especifico.

Funciones de Orden Superior: se llaman funciones de orden superior a las que:

1. Reciben una función como parámetro.
2. Devuelven otra función como resultado.
3. hacen las dos cosas a la vez.

Texto

El contenido generado por IA puede ser incorrecto.Texto

El contenido generado por IA puede ser incorrecto.Texto

El contenido generado por IA puede ser incorrecto.En vez de trabajar solo con datos (números, strings, objetos), también trabajan con funciones como si fueran datos.

Interfaces Funcionales: La principal dificultad que se evidencia en un lenguaje orientado a objetos y fuertemente tipado como es Java radica en que su sintaxis no es flexible para un dato de tipo función. Al ser orientado a objetos, se debe trabajar con clases, objetos y métodos. Java logra ofrecer variables de tipo función mediante el concepto de las interfaces funcionales. Una interfaz funcional es una interfaz que posee un único método, la cual puede estar opcionalmente anotada como @FunctionalInterface. Esta anotación no impide su uso para programación funcional, pero previene del potencial error de que se le agregue un segundo método.

Funciones lambda: Una expresión lambda tiene la siguiente sintaxis (parámetros) -> { cuerpo }

Dado que una interfaz funcional posee un único método, resulta redundante programar una nueva clase con una sobreescritura. Si se declara una variable de referencia a dicha interfaz, es inevitable que debe referenciar a una nueva implementación que sobreescriba es único método. Las expresiones lambda ofrecen la posibilidad de escribir únicamente el bloque de código del método directamente en el lugar de la declaracion de la variable, el cual es considerado automáticamente como la sobreescritura del método. Las expresiones lambda pueden omitir los paréntesis en el caso de poseer un único parámetro. Y de la misma manera si su cuerpo consiste en una única instrucción return pueden ser omitidas las llaves y la palabra clave return.

Interfaces funcionales provistas

Las diversas interfaces funcionales que provee el paquete java.util.function se diferencian entre sí únicamente por la cantidad y tipo de parámetros y el tipo de retorno. Existe una cantidad bastante grande de interfaces funcionales provistas, para diferentes combinaciones de tipos de parámetros y de retorno. La más destacas son:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Stream API: La API de Streams permite procesar datos como un flujo continuo en lugar de como una colección almacenada completa en memoria. A diferencia de las colecciones que se recorren con bucles, los streams hacen transitar cada elemento por una “tubería” de operaciones, liberando al programador del control del recorrido. La analogía de la cascada ilustra que cada dato (gota) puede ser interceptado por operaciones intermedias antes de llegar al resultado final.

Los flujos pueden originarse desde colecciones (Collection.stream()), arreglos (Arrays.stream()), valores literales (Stream.of()), o como flujos infinitos mediante Stream.iterate(inicial, unaryOperator) y Stream.generate(supplier), que requieren acotarse con limit(). Existen además generadores específicos según el dominio, como Files.lines() para leer líneas de archivos o getResultStream() en JPA para recuperar entidades sin materializarlas en listas.

Las operaciones intermedias transforman o filtran cada elemento y devuelven nuevos streams, permitiendo el encadenamiento ordenado de múltiples pasos. Por lo general, dichas operaciones reciben **interfaces funcionales** (p. ej., Function, Predicate, Supplier, Consumer, UnaryOperator), que suelen implementarse mediante **lambdas** para expresar la lógica de forma concisa y declarativa.

**filter**  
Aplica un Predicate<T> a cada elemento del flujo y conserva solo aquellos para los que la condición resulta verdadera, devolviendo un nuevo stream con los filtrados. Útil para seleccionar subconjuntos de datos.

**distinct**  
Elimina duplicados del flujo y retorna uno nuevo con elementos únicos. La unicidad se determina por equals()/hashCode() de los objetos, por lo que depende de su correcta implementación.

**map**  
Transforma cada elemento del flujo aplicando una Function<T,R>, produciendo un nuevo stream cuyos elementos pueden ser de **otro tipo**. Se utiliza para proyecciones y conversiones.

**limit**  
Restringe el flujo a como máximo *n* elementos, devolviendo un stream truncado. Es clave para **acotar flujos infinitos** generados con iterate() o generate().

**sorted**  
Devuelve un nuevo flujo con los elementos ordenados. Si los elementos implementan Comparable, usa el orden natural; alternativamente acepta un Comparator para definir el criterio de ordenación.

**Operaciones terminales**

Señala que un stream **solo se procesa** cuando se invoca una **operación terminal**; hasta ese punto, las operaciones intermedias permanecen perezosas. Las terminales consumen el flujo y **no devuelven otro stream** (devuelven un valor, una colección o nada).

**Más usadas**

* **count**: devuelve la cantidad de elementos del flujo.
* **forEach**: ejecuta un Consumer por cada elemento; no retorna resultado.
* **min / max**: retorna el mínimo o máximo según el **orden natural** (Comparable) o un **Comparator** provisto.
* **anyMatch / allMatch / noneMatch**: reciben un Predicate y retornan boolean. Verifican respectivamente: si **algún** elemento cumple, si **todos** cumplen o si **ninguno** cumple.

**Collectors**

Define que un **Collector** especifica cómo **acumular, combinar y finalizar** los datos de un stream para producir un **resultado final**. Se usa con collect(...), y la clase Collectors ofrece implementaciones comunes.

**Colecciones y estructuras**

* **toList / toSet / toCollection**:
  + toList() produce una lista (desde Java 16, típicamente **no modificable**).
  + toSet() elimina duplicados (sin orden garantizado).
  + toCollection(supplier) permite elegir la estructura (p. ej., ArrayDeque::new).

**Strings**

* **joining(delimitador[, prefijo, sufijo])**: concatena los elementos (p. ej., "Ana, Bruno, Ana"), con opciones de prefijo/sufijo.

**Agrupaciones y particiones**

* **groupingBy(clave)**: agrupa en Map<Clave, List<T>>.
  + **Downstream collectors**: combinables para contar (counting()), transformar (mapping(...)), o resumir (p. ej., averagingInt(...)).
* **partitioningBy(predicado)**: divide en dos grupos Map<Boolean, List<T>> según el Predicate.

**Mapas**

* **toMap(clave, valor, merge[, mapSupplier])**: construye mapas controlando resolución de **claves duplicadas** con una función de **merge** (p. ej., Integer::max) y el tipo de mapa (p. ej., LinkedHashMap::new para mantener orden de inserción).

**Estadísticas rápidas**

* **summarizingInt / Long / Double**: devuelve \*SummaryStatistics con **count, min, average, max** (y suma en los tipos que la exponen).

**reduce (pliegue del stream)**

Explica que reduce aplica repetidamente un **BinaryOperator** para combinar elementos hasta obtener **un único resultado** (p. ej., suma acumulada).

**Variantes de reduce**

1. **Sin identidad**: reduce(BinaryOperator) → retorna Optional<T> (el stream podría estar vacío).
2. **Con identidad**: reduce(identidad, acumulador) → evita Optional y define un valor inicial seguro (ej.: 0 para suma, 1 para producto).
3. **Con identidad, acumulador y combinador**: reduce(identidad, acumulador, combinador) → pensada para **paralelismo**; el **acumulador** procesa parciales y el **combinador** fusiona resultados intermedios.

**Referencias a métodos (resumen en 3ª persona)**

Explica que las **method references** son una forma concisa de pasar funciones ya existentes cuando la firma coincide con la interfaz funcional esperada (no ejecutan en el momento, solo referencian).

**Tipos:**

* **Método estático:** Clase::metodoEstatico Ej.: Math::abs, Integer::parseInt.
* **Método de instancia de un objeto concreto:** objeto::método Ej.: scanner::nextLine.
* **Método de instancia de un tipo arbitrario:** Tipo::método Ej.: String::toUpperCase (se aplicará al elemento del stream).
* **Constructor:** Tipo::new Ej.: Persona::new (para crear instancias en map).

**Ejemplos breves:**

list.forEach(System.out::println); // Consumer<String>

Stream.of("1","2").map(Integer::parseInt); // Function<String,Integer>

Files.lines(path).map(Persona::new); // Function<String,Persona>

**Streams primitivos (IntStream, LongStream, DoubleStream)**

Señala que existen para **evitar boxing/unboxing** y mejorar rendimiento en pipelines numéricos largos.

**Creación típica:**

IntStream.range(1, 10); // 1..9

IntStream.rangeClosed(1, 10); // 1..10

new Random().ints(5); // 5 enteros aleatorios

**Terminales útiles:** sum(), average(), min(), max(), summaryStatistics().

**Conversiones:**

List<Integer> xs = List.of(1,2,3);

IntStream s = xs.stream().mapToInt(Integer::intValue); // a primitivo

List<Integer> ys = s.boxed().toList(); // a objetos

**Regla práctica:** si el pipeline es **numérico y largo**, conviene usar streams primitivos; si se trabaja con **objetos** (p. ej., Persona), se prioriza claridad y se convierte a primitivo **solo** donde aporta.

**Buenas prácticas y anti-patrones**

* **Ejecución perezosa y uso único:** las intermedias no corren hasta una **terminal**; un stream **no** se reutiliza.
* **Evitar side-effects:** preferir map/filter puros; si se necesita salida, recolectar con collect(...) en lugar de mutar colecciones externas.
* **peek solo para depurar:** no usarlo para lógica de negocio.
* **reduce con identidad y combinador:** definir identidad correcta (0 suma, 1 producto) y combinador para paralelismo.
* **I/O con Files.lines(...):** usar **try-with-resources** y charset adecuado; el stream cierra el archivo al salir del try.
* **Cuándo NO usar Streams:** cuando un for simple es más claro/rápido, se requiere control de flujo complejo, o se mutan estructuras compartidas intensamente.

**Checklist rápido:**

1. ¿Es más legible que con for?
2. ¿Sin side-effects en el pipeline?
3. ¿Recursos cerrados correctamente?
4. ¿Hace falta recolectar o alcanza una terminal (forEach, count)?

**flatMap (idea y ejemplos)**

Aclara que **aplana** streams de colecciones en un solo stream de elementos.

// Tokenizar frases

List<String> tokens = frases.stream()

.flatMap(f -> Arrays.stream(f.split(" ")))

.toList();

// Listar alumnos de todos los cursos

List<String> todos = cursos.stream()

.flatMap(c -> c.alumnos().stream())

.toList();